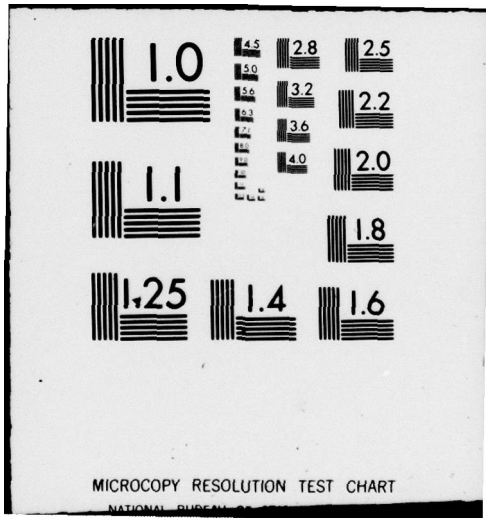
AD-A080 128 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
THE POST PROCESSING FUNCTIONS OF A DATABASE COMPUTER. (U)
JUL 79 D K HSIAO, J MENON N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-79-6 NL

| OF |
AD
A080128

END
DATE
FILMED
4-80
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART

TECHNICAL REPORT SERIES

LEVEL II

12

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

(OSU-CISRC-TR-79-6)

The Post Processing Functions of a Database Computer

by

David K. Hsiao

and

Jaishankar Menon

Work performed under

Contract N00014-75-C-0573

Office of Naval Research

Computer and Information Science Research Center

The Ohio State University

Columbus, OH 43210

July 1979

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: OSU-CISRC-TR-79-6	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): The Post Processing Functions of a Database Computer	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): David K. Hsiao, Jaishankar Menon	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Washington, D. C. 20360	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ONR 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: July 1979
	13. NUMBER OF PAGES: 32
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicago
ONR 102IP	ONR, Pasadena

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By letter on file
Distribution/
Availability Codes
Dist. | Avail and/or special
A

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database computer, post processing, implicit join, natural join, source records, target records, processor interconnections, set functions.

10 TO THE 10TH POWER

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DBC is a specialized back-end computer which is capable of managing database of 10^{10} bytes in size and supporting known data models such as relational, network, hierarchical and attribute-based. This report deals with the post processing functions of DBC. A description of some known methods for performing natural and implicit joins is first given. We then go on to show how both natural and implicit joins are performed by the post processor (PP) of DBC utilizing the parallelism of PP. We also show that the algorithm

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407586

for performing joins is of O(N) time, where N is the number of records to be joined.

Algorithms necessary for performing set functions such as maxima, minima, average, sum and count are given. The time complexities of these algorithms are also calculated.

Finally, it is shown how to implement the set inclusion operator. Given a set of values for a particular attribute and a number of retrieved records, this operator can select those records whose values for the attribute are the values in the set.

PREFACE

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Professor of Computer and Information Science, and conducted at the Computer and Information Science Research Center of The Ohio State University. The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report is based on research accomplished in cooperation with the Department of Computer and Information Science. The research contract was administered and monitored by The Ohio State University Research Foundation.

TABLE OF CONTENTS

1. INTRODUCTION

In the previous technical reports[1-3], we presented various aspects of the design of a database computer known as DBC. The architecture of DBC is shown in Figure 1. This report deals with the post processing functions of DBC. These functions are carried out by the part of the security filter processor (SFP), known as the post processor (PP). The other part of SFP, i.e., the security and clustering unit (SCU), was documented in a previous report [4]. We shall not repeat the discussion of SCU here. This report is solely devoted to a discussion of PP. The functions of the post processor (PP) are considered in the following three categories:

(1) Sorting of retrieved records. Once the response set of a user query is given to PP by the mass memory, this set of records can be sorted in the post processor.

(2) The natural and implicit join operations on two sets of records retrieved from the mass memory. These operations are particularly needed on records retrieved from relational databases.

(3) The set functions (maxima, minima, average, count, sum, set inclusion) on the response set of a user query after the records of the response set have been retrieved from the mass memory (MM) and given to the security filter processor (SFP).

The advantage of performing these operations in a separate unit, namely, the post processor (PP), is twofold. First, it enhances the pipelining achievable in DBC. Thus, post processing functions may be performed by PP on the response set of one query in conjunction with the evaluation of another query by MM. Second, it preserves the functional specialization of each component of DBC and improves the overall reliability of DBC, since MM

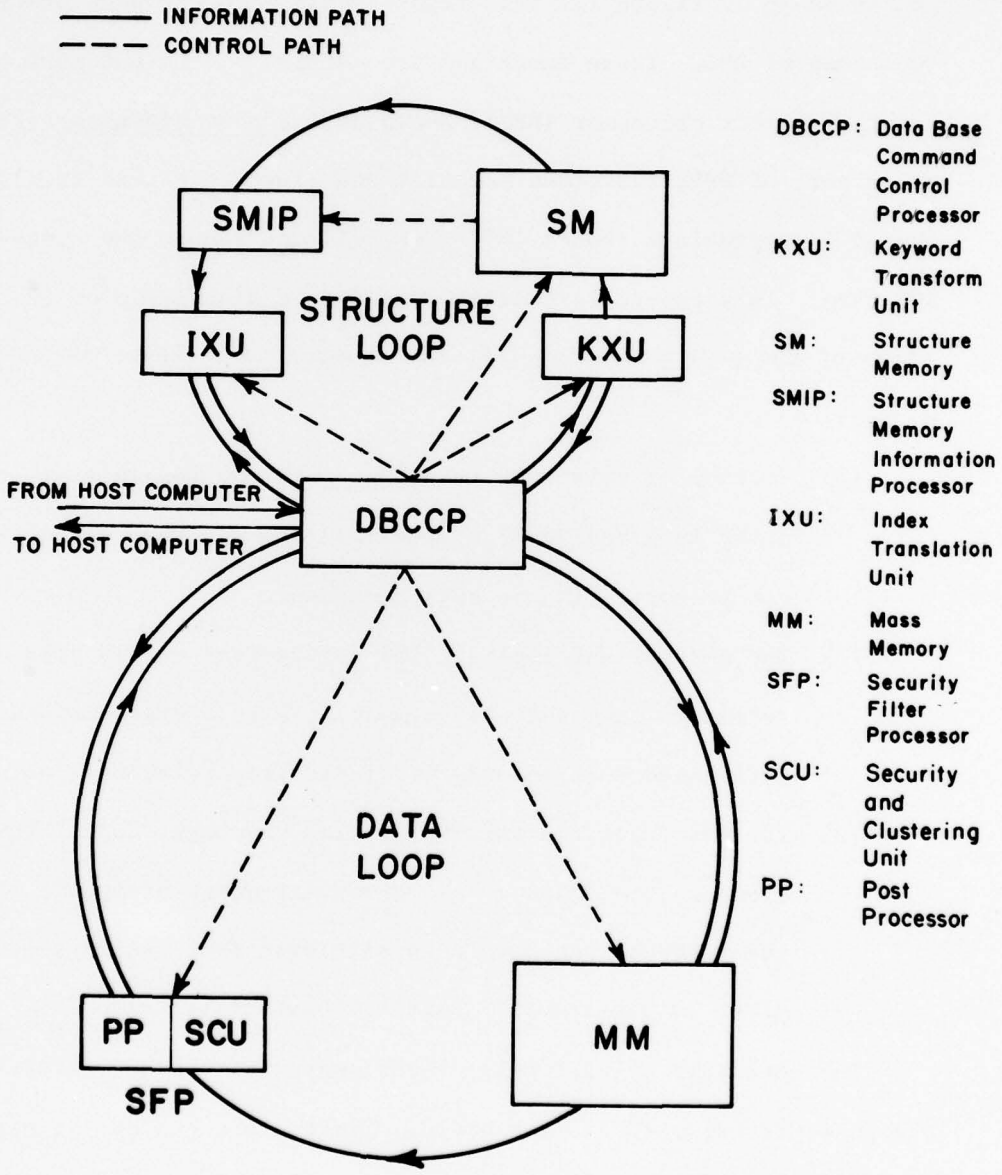


FIGURE 1 The Architecture of DBC

and PP are now both fairly simple components.

We have already discussed in [5], how we intend to perform the sorting operation in PP. Thus, we shall concentrate on the discussion of the remaining two categories of post processing operations. As a parallel processing unit, PP is configured and interconnected with many identical processors and local memories as depicted in Figure 2. We note that PP consists of n processors (number 0 through n-1), each having sequential memory of size m where n is a power of 2 and m is a positive integer. It is important to see that

(1) Each processor has a direct connection to log n other processors.

(2) There is a sequence of all the processors $p_1, p_2, \ldots, p_n$, such that p_i is connected to p_{i+1} for all $1 \leq i \leq n-1$, and $p_i \neq p_j$ for all $1 \leq i \leq n$ and all $1 \leq j \leq n$. We say that p_{i+1} follows p_i in the sequence. Also, p_n is connected to p_1 and we say p_1 follows p_n in the sequence.

We shall call such a sequence of processors a broadcast sequence. For example, in the case where n=16, the broadcast sequence is 0,1,3,7,5,13,15, 11,9,8,10,14,12,4,6,2. We may easily see that the broadcase sequence is merely the sequence of processors along the periphery of the polygon drawn to represent the interconnection of processors. PP has a post processing controller (PPC) to coordinate the work of the various processors in PP. PPC is directly connected to all the processors and contains a small associative memory to speed up the join operations. These operations are discussed in the next section.

2. THE JOIN OPERATION

We may recall that information is stored into and retrieved from DBC in terms of records; a record is made up of a collection of keywords and a record body. The record body consists of a (possibly empty) string of characters which are not used for search purposes. A keyword is an attribute-value pair,

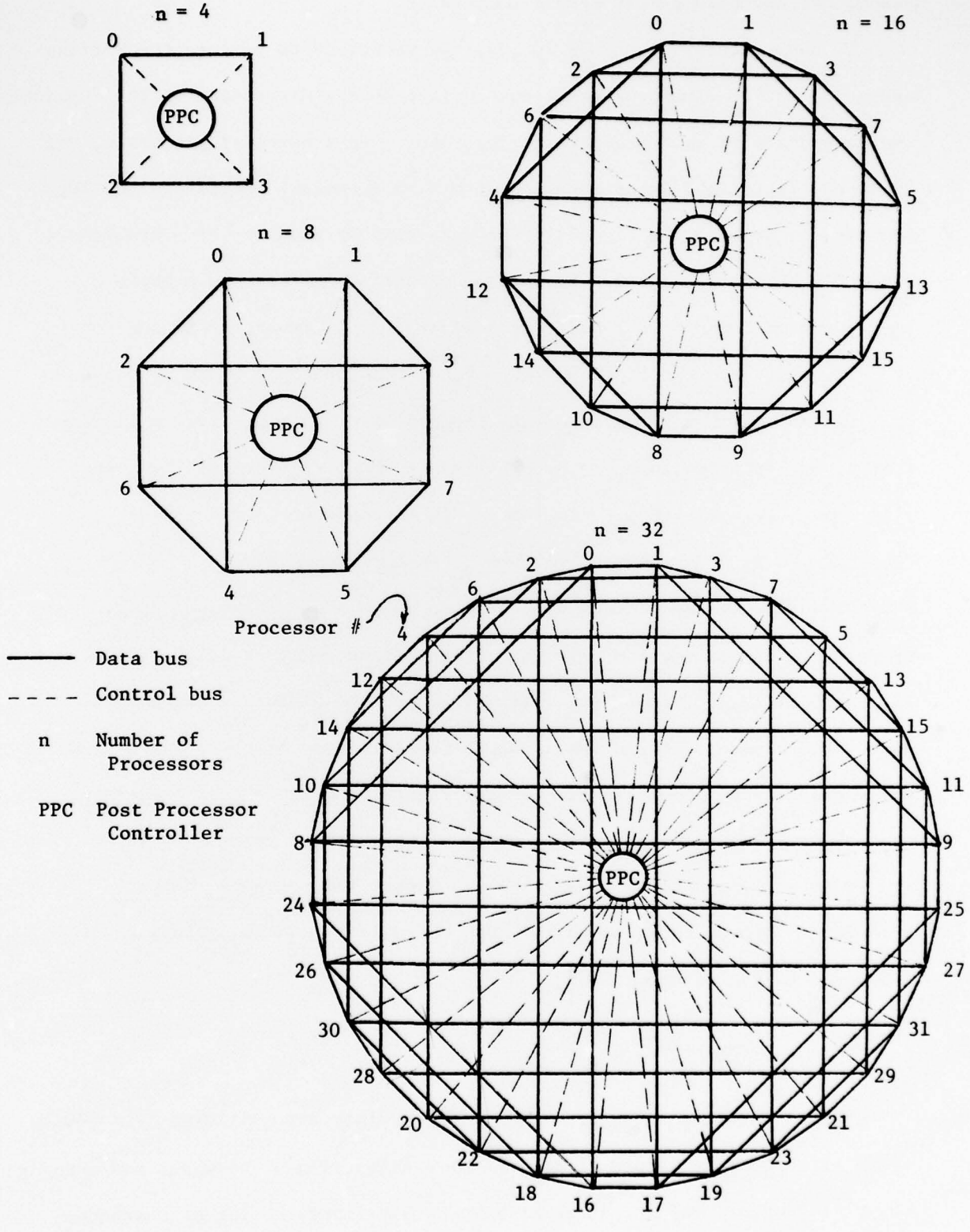


Figure 2. Interconnection of Processors for Post Processing

where the attribute may represent the type, quality or characteristic of the value. An example of a record with three keywords is shown below:

(<Relation, EMPLOYEE>, <Salary, 5000>, <Job, MANAGER>).

The above record has three keywords. The list of attributes constituting the above record is [Relation, Salary, Job].

Two sets of records are involved in a join operation. For a record set to participate in a join operation, it must have the following special property. Each record in the set must have the same list of attributes. Thus, a record set is characterized by a list of attributes. The following set of two records possesses this property.

Record 1: (<Job, MANAGER>, <Name, HSIAO>).

Record 2: (<Job, SECRETARY>, <Name, ANNE>).

The list of attributes that characterizes the above set is [Job, Name]. Each attribute in the list of attributes that characterizes a record set is said to belong to that set. Thus, in the above example, Job and Name are attributes that belong to the set that they characterize. One of the sets that participates in the join operation is called the source set. The other set is called the target set. Records in a source set are called source records and records that make up the target set are called target records. A join operation is always performed between an attribute that belongs to the source set and an attribute that belongs to the target set. We call these two attributes as the source attribute and the target attribute, respectively. Furthermore, a join operation requires that domain of values that the source attribute may take must be exactly the same as the domain of values that the target attribute may take.

For example, consider that the source set is characterized by the list of attributes [Name, Salary], and that the target set is characterized by the list of attributes [Salary, Department-Number]. Now, assuming that the domain of the Name attribute consists of some specific alphabetical strings and the domains of the Salary and Department-Number attributes consist of some specific numbers. Therefore, a join operation may be performed between the Salary attribute belonging to the source set and the Salary attribute belonging to the target set (i.e., we choose Salary as both the source and target attributes). Similarly, another join operation may be performed between the Salary attribute belonging to the source set and the Department-Number attribute belonging to the target set (i.e., we choose Salary as the source attribute and Department-Number as the target attribute). However, no join operation is possible if we choose Name as the source attribute, because no attribute belonging to the target set has the same domain as the Name attribute.

A join operation is performed as follows. Each record in the source set is examined and the value of the source attribute is determined. All records in the target set which have the same value for the target attribute are now selected. If the user has requested only the values of attributes from the selected target records, the resulting operation by the PP is called an implicit join. However, if the user has requested the values of attributes from the selected target records as well as of attributes from the source records that were used to make the selection, the resulting operation is a natural join.

We shall clarify the above concepts by means of an example. Consider that the source set is characterized by the list of attributes [Name, Salary], and that the target set is characterized by the list of attributes [Salary, Department-Number]. There are two join operations as follows:

(1) Join the source set and the target set using Salary as both the source and target attributes, and extract the values of the Department-Number attribute.

(2) Join the source set and the target set using Salary as the source and target attributes, and extract the values of the Name and Department-Number attributes.

We note that the first join operation is implicit and the second join operation is natural.

2.1 A Known Method for Implicit Join

In this section, we shall consider a method for implicit join that is used in the database machine CASSM [6, 7]. CASSM has a cellular architecture in which each cell incorporates a special-purpose microprocessor and uses a circulating sequential memory like the track of a disk. Associated with each cell, there is a 1-bit-wide RAM. To support the *implicit join operation*, a cell is allowed to access the RAM associated with any other cell (possibly indirectly, via one or more other cells). Attribute values are stored by the their coded values. A unique coded value is assigned to each distinct attribute value and the coded values fall in the range between 1 and the number of all distinct values of an attribute's domain. Now, the RAM's of all cells are pooled. Each bit of the pooled RAM corresponds to a unique value of a data field. The situation is shown in Figure 3.

The join operation proceeds as follows. Each record of the source set is examined and the RAM bit corresponding to the coded value of the source attribute is marked. The records of the target set are now examined one at a time. If the RAM bit corresponding to the coded value of the target attribute has been marked, then this record is selected and the necessary information is obtained from it.

P_i: the i-th logic-per-track processor

hxn: the (maximum) number of unique values of an attribute

----: processor-memory connection for pooling purposes

——: processor-memory connection for all purposes

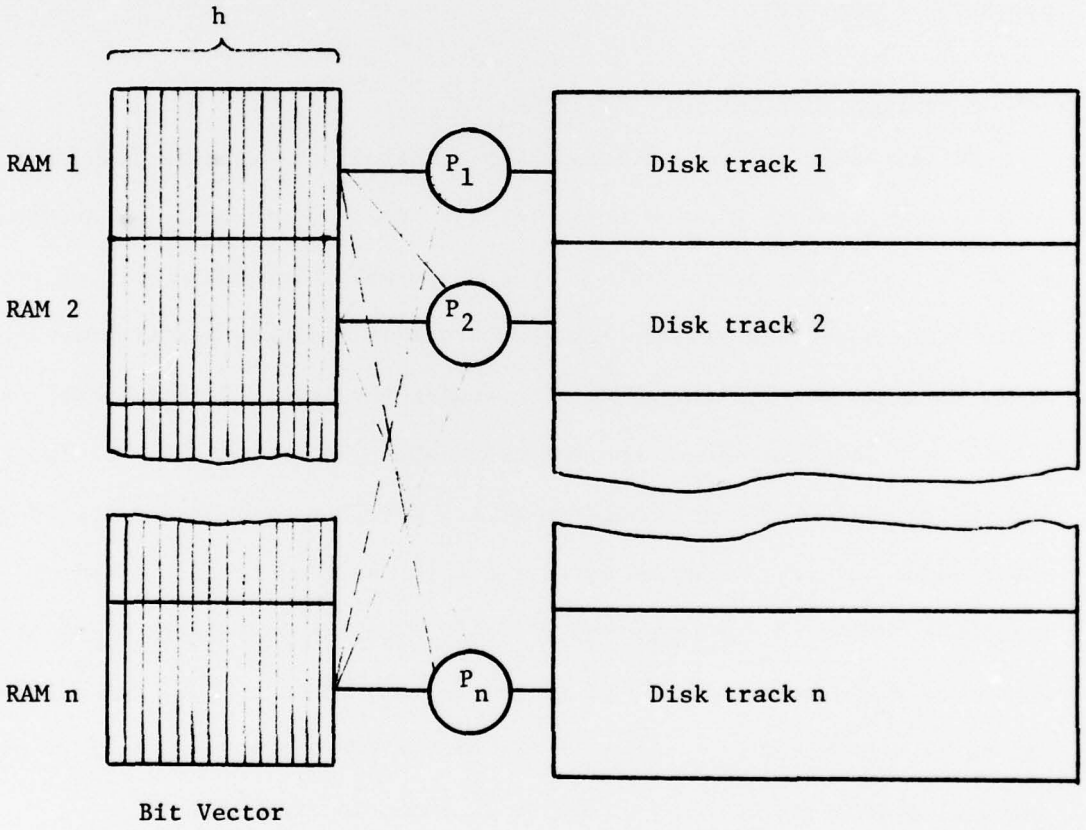


Figure 3. The CASSM Architecture Supporting the Implicit Join Operation

Notice, that we will not be able to perform a natural join using this method. This is because all the source records are first examined. The target records are examined next and a record is selected if the RAM bit corresponding to the value of its target attribute has been marked. However, information regarding which record in the source set caused the marking of this RAM bit is lost. Hence, values of attributes from this record in the source set cannot be obtained.

Besides the fact that this method cannot be used to perform a natural join, there are other limitations. First, all data values have to be uniquely coded. Second, since each processor may have to access the RAM associated with every processor, n^2 interprocessor connections may be required, where n is the number of processors. Otherwise, a processor which needs to access the RAM of another processor may encounter a serial delay.

In DBC, we do not use any coding to store data values in the mass memories (MM). Also, each processor in PP is only connected to log n other processors. Thus, this method is unsuitable for use in DBC.

2.2 A Known Method for Natural Join

The natural join, as we described it in an earlier section, is essentially an (n1×n2) operation, where n1 is the number of records in the source set and n2 is the number of records in the target set. This is because, for each of the n1 records in the source set, all the n2 records of the target set have to be examined.

The following method which is often used requires only (n1+n2) operations rather than (n1×n2) operations. The records of the source set are sorted on the source attribute and the records of the target set are sorted on the target attribute. The join operation now proceeds as follows. The first source record is examined and the source attribute value is noted. The target records are now examined for a possible match until a record is found

whose target attribute value is greater than the source attribute value previously noted. No more target records have to be examined since their target attribute values are also greater than the source attribute value of the source record.

The advantage of this method is that both natural and implicit joins may be performed. However, the need to sort the sets initially makes it unattractive.

2.3 A New Method for Natural Join

In DBC, we use a method that requires only (n1+n2) read operations distributed among n processors, but does not require the sorting of the two sets initially.

We would like to load the records in such a way that they are evenly distributed among the memories of the processors. Thus, consider that there are 20 processors each with memory capable of holding 20 records. Then, we would like a set of 40 records to be placed two records per processor-memory rather than in two processor-memories alone.

In order to assure this even distribution of records in the post processor (PP), we propose the interface shown in Figure 4. The outputs from the track information processors (TIPs) of the mass memory are merged into a single stream. This may be done by collecting all the records in a single buffer and sending them out, one record at a time at each clock pulse. The output from this single stream is fed into a demultiplexer. The demultiplexer selects each of the n possible outputs in a round robin fashion. This may be easily arranged by letting the input to the selection lines of the demultiplexer be the output of a modulo-n counter that changes its state with the arrival of a record, i.e., at each clock pulse. Thus, we have arranged that the processor memories will be evenly filled up with records. This will improve the perfor-

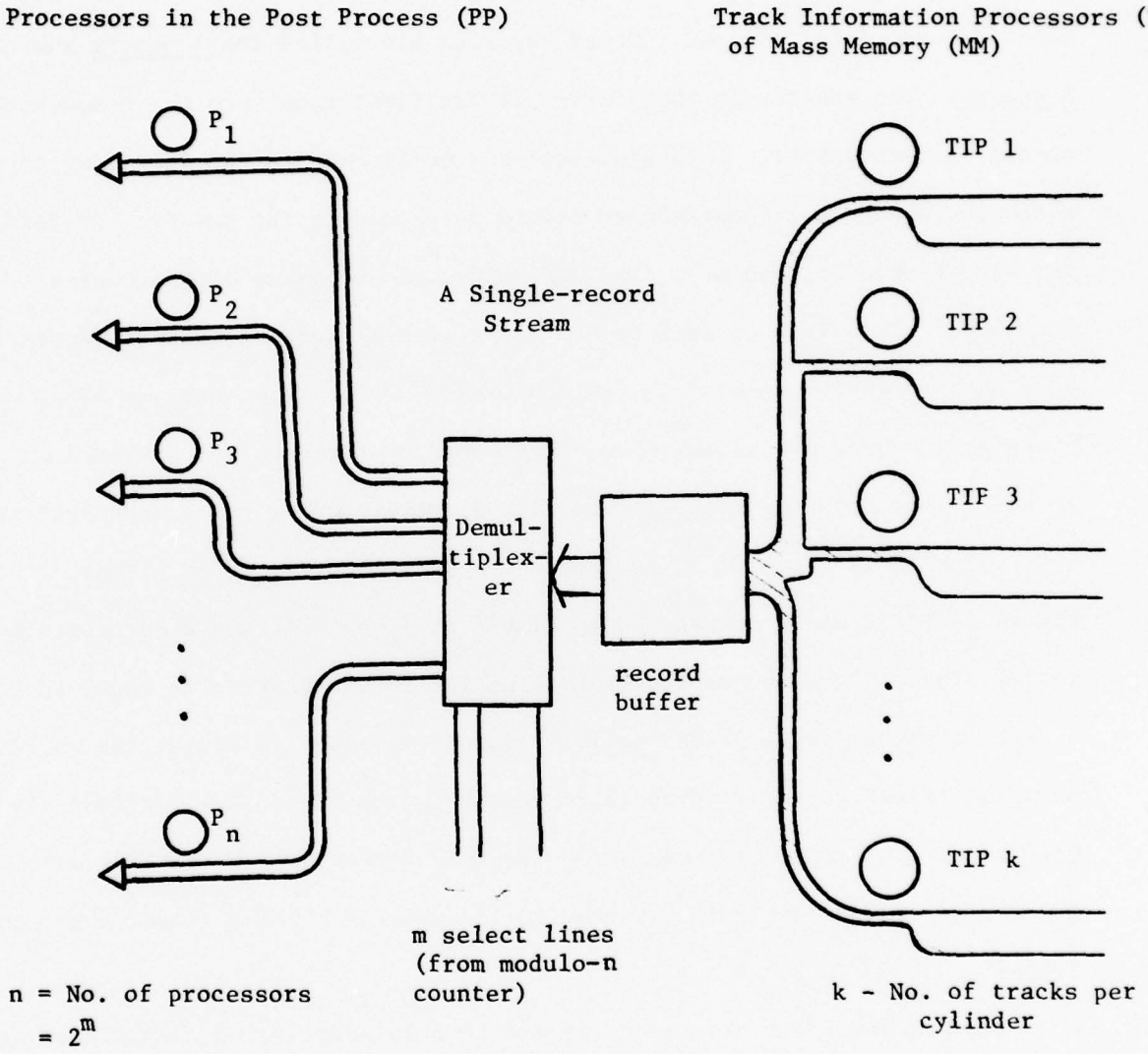


Figure 4. MM-PP Interface Used to Ensure Even distribution of Records to Processor Memories

mance of PPC in doing its various operations since all the processors will have the work equally divided among themselves.

We require, for this method, that each processor have two sequential memories associated with it. These memories are called the A-memory and the B-memory. The records in the source set are first read into the A-memories of all the processors. Each processor begins to execute the following algorithm the moment the first source record is placed in its memory. It reads the source records, one at a time, and extracts the value of the source attribute. This is then sent to PPC along with a 'request-list-#' request.

An associative memory (AM) is employed in PPC. Each entry in AM contains a source attribute value. Each source attribute value stored in AM has a corresponding list-#. The list-# corresponding to a source attribute value SA is i, if SA is stored in the i-th location of AM. Thus, in Figure 5, HSIAO has a corresponding list-# of 1, and JAI has a corresponding list-# of 2. PPC searches its AM looking for an entry which is equal to the source attribute value passed to it. If such an entry is found, the corresponding list-# is returned to the processor which issued the 'request-list-#'. Otherwise, a new entry is made in AM for this source attribute value and the corresponding list-# is returned to the processor which issued the 'request-list-#'.

The processor now hashes the list-# into an address. A simple hashing function would be

$$\text{address} = \text{list-\#} \bmod m$$

where m is the size of a processor memory. The address refers not to a single memory location, but to a certain block of sequential memory, i.e., B-memory, wherein many records may be stored. The record is then stored in the block of the B-memory, if space is available in this block. Otherwise, it is stored in an overflow area. A few blocks in each processor memory are reserved to store overflow

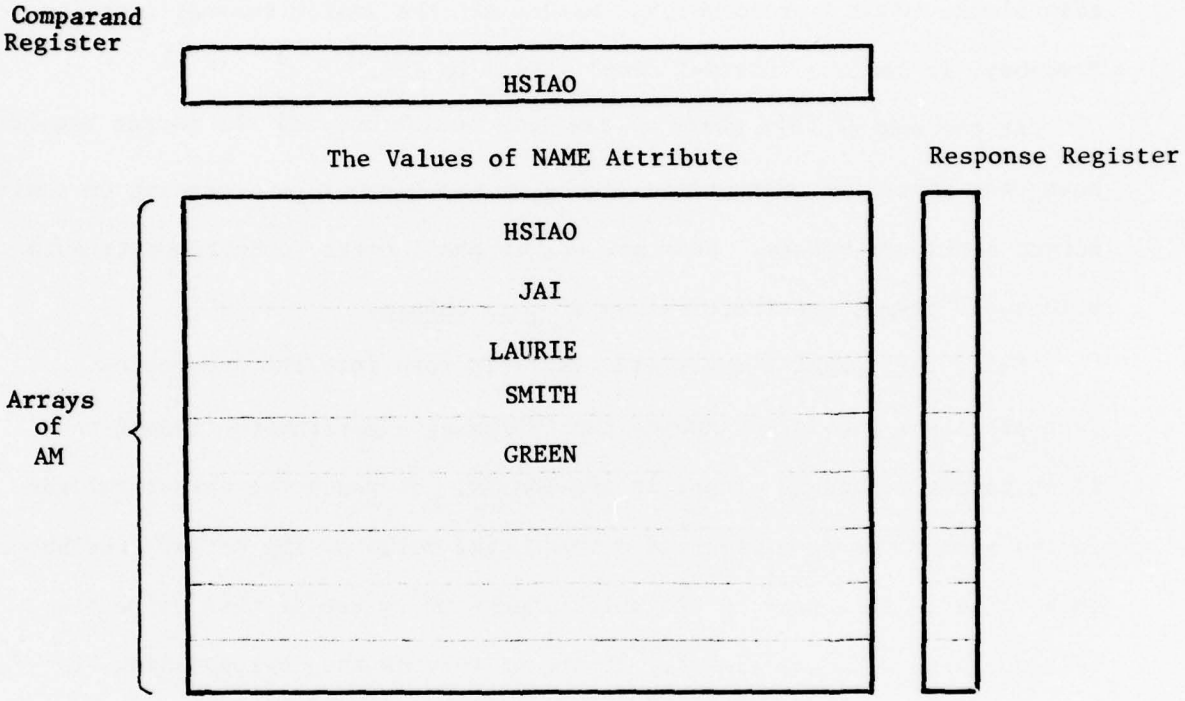


Figure 5. Logical View of the Associative Memory (AM) of PPC Showing How the Value 'HSIAO' is searched for.

records. Each record in the overflow area contains a pointer to the next record in the overflow area which hashed to the same block. One memory location in each block is reserved for an overflow pointer. The overflow pointer contains a pointer to the first overflow record in the overflow area that hashed to this block. Once a processor has hashed all the source records into its B-memory, it sends a 'phase-I-over' signal to PPC.

At the end of this phase of the join operation, all the source records have been placed in appropriate blocks of the B-memories depending on their source attribute values. From now on, we shall refer to source attribute values and target attribute values as join values.

Next, the target records are similarly read into the A-memories. Each processor begins to execute the following algorithm the moment the first target record is placed in its memory. It reads the target records in its memory one at a time and extracts the value of the target attribute. This value is then sent to PPC which checks AM to see if this value can be associated with any list-#. If so, it returns the corresponding list-#. Otherwise, it returns a null value for the list-#. The processor then hashes the non-null list-# to a block address. The hash algorithm used here is the same one that was used earlier to hash the join values of source records. All the source records in this block of the B-memory and all overflow records belonging to this block are now looked at. If the join value is the same as the join value of the target record that was hashed to this block, then this source record will participate in the join operation. The necessary attribute values are extracted from both the source and target record and sent to DBCCP which then routes them to the front-end computer from which the request originated.

If the list-# corresponding to the join value of the target record was non-null, this record and the list-# corresponding to its join value are

propagated to all other processors via the broadcast sequence already discussed in an earlier section. That is, each processor sends the record to one other processor -- the one following it in the broadcast sequence. In this way, the record is propagated to every other processor.

Every processor that obtains a broadcasted record and a list-#, does the following. The list-# is hashed to a block address in its B-memory. All records in this block and all overflow records belonging to this block are examined for possible participation in the join operation. We see that the need to examine these records for possible participation in the join operation arises only because the hashing function may not be one-to-one, so that two different list-#'s may hash to the same address.

Once a processor has examined all target records in its A-memory, it sends a 'phase-II-over' signal to PPC.

2.3.1 Discussion of Interprocessor Communication

We may recall, that each processor is directly connected to PPC. Since PPC can handle only one request at a time, it has a Condition Flag. A processor will issue a request to PPC only if the Condition Flag shows that PPC is not busy. Otherwise, it will hold the request until PPC becomes free. PPC marks this flag 'busy' in acceptance of a request and marks it 'free' when it finishes responding to a request.

It is also necessary for a processor to send records to one other processor. Each processor owns a register that can hold one record. This is called the Record Register. Associated with each Record Register is a Flag Bit which indicates whether the Record Register contains a record or not. A processor p_i sends a record over to another processor p_j only if the Flag Bit of p_j indicates that the Record Register of p_j is empty. The Flag Bit of a processor is marked by the processor that sends records over to it. The Flag Bit of a processor is cleared by itself. A record received from another processor is

first hashed to a block in the B-memory. All records in this block and all overflow records belonging to this block are now examined for possible participation in the join operation. After this, the record is broadcast to the next processor in sequence, if need be.

The format of the Record Register is shown in Figure 6. It consists of three parts. The first part holds the record itself. The second part contains the list-# corresponding to the join value of the record and the third part contains a Token. The initial value of this Token is one. Each processor that receives the record increments the value of the Token by one. If the value of the Token is less than n, the record and Token are passed along to the next processor in the broadcast sequence. Otherwise, the record is discarded since all the processors have already obtained this record. Whenever processor i sends a record to processor j, it marks processor j's Flag Bit. After processor j has examined this record for participation in the join operation and passed it along to the next processor, if need be, it clears its own Flag Bit to indicate its readiness to accept other records.

The Record Register is essentially a message buffer of size one. We feel that the buffer only needs to be of size one, since the processors are capable of processing the records and sending them on to the next processor, if need be, very quickly.

2.3.2 Algorithms Executed by PPC

In order to execute a join operation, two sets of records must be retrieved from the mass memory (MM) and placed in the post processor (PP). The entire execution is controlled by the database command and control processor (DBCCP) in the following manner. First, it issues a 'retrieve-by-query' request to MM in order to retrieve the source records. At the same time, it issues a 'begin-phase-I-of-join' command to PPC of PP. Later, it issues a 'retrieve-by-query' request to MM in order to retrieve the target records. Simultane-

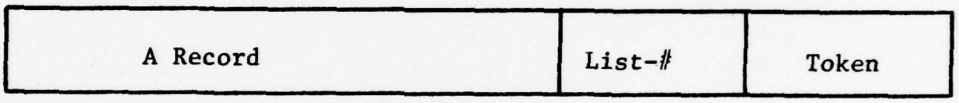


Figure 6. A View of the Record Register
Used for Interprocessor Communication

ously, it issues a 'begin-phase-II-of-join' command to PPC.

PPC executes the following algorithms.

Algorithm 1:

Executed in response to: 'begin-phase-I-of-join' command from DBCCP.

Input arguments: None

Step 1: Condition Flag='Busy'.
Step 2: Issue 'begin-phase-I' request to all the n processors.
Step 3: List=0 [List is used by Algorithm 2 to assign list-#'s]. Phase-I=0 [A variable that stores the number of processors that have finished Phase-I of the join operation]. Phase-II=0 [A variable that indicates the number of processors that have finished Phase-II of the join operation].
Step 4: Condition Flag='Free'.
Step 5: Terminate.

Algorithm 2:

Executed in response to: 'request-list-#' command issued by one of the processors.

Input arguments: (a) The number N of the processor making the request.
(b) A join value.

Step 1: Condition Flag='Busy'.
Step 2: Return 'request-accepted' signal to processor N (Argument a) that issued the request.
Step 3: Search AM looking for an entry which is equal to the join value (Argument b). [Request is a variable that indicates whether DBCCP has already issued the 'begin-phase-II-of-join' command.] If such an entry is found, go to Step 5. Else, if Request=1, then return a null list-# and go to Step 6. Otherwise, go to Step 4.
Step 4: List=List+1. Make a new entry in AM of PPC, where this entry will contain the join value (Argument b). Return the value of List to processor N which made the request. Go to Step 6.
Step 5: Return the corresponding list-# to processor N which made the request.
Step 6: Condition Flag='Free'.
Step 7: Terminate.

Algorithm 3:

Executed in response to: 'phase-I-over' command from one of the processors.

Input arguments: None.

Step 1: Condition Flag='Busy'.
Step 2: Phase-I=Phase-I+1. If Phase-I=n and Request=1, then execute Algorithm 6.
Step 3: Condition Flag='Free'.
Step 4: Terminate.

Algorithm 4:

Executed in response to: 'phase-II-over' command from one of the processors.

Input arguments: None.

Step 1: Condition Flag='Busy'.
Step 2: Phase-II=Phase-II+1. If the Phase-II=n, then execute Algorithm 7.
Step 3: Condition Flag='Free'.
Step 4: Terminate.

Algorithm 5:

Executed in response to: 'begin-phase-II-of-join' command from DBCCP.

Input arguments: None.

Step 1: Condition Flag='Busy'.
Step 2: Set Request=1. If Phase-I=n, then execute Algorithm 6.
Step 3: Condition-Flag='Free'.
Step 4: Terminate

Algorithm 6:

Executed in response to: A call from either Algorithm 3 or Algorithm 5. Basically, it is executed after all processors have finished Phase-I of the join and DBCCP has issued the 'begin-phase-II-of-join' command.

Input arguments: None.

Step 1: Condition Flag='Busy'.
Step 2: Issue 'begin-phase-II' request to all the n processors.
Step 3: Condition Flag='Free'.
Step 4: Terminate.

Algorithm 7:

Executed in response to: Call from Algorithm 4.

Input arguments: None.

Step 1: Condition Flag='Busy'.
Step 2: I=0
Step 3: Examine Flag Bit of processor I. If Flag Bit=1, then go to Step 2. Else, I=1+1. If I=n, then go to Step 4. Else, go to Step 3.
Step 4: Send 'Stop' signals to all processors.
Step 5: Condition Flag='Free'.
Step 6: Terminate the join operation.

The above algorithm is a polling algorithm that checks to ensure that the Flag Bits of all processors are 0. This indicates that no more routing of records has to be performed and that the join operation is over.

2.3.3 Algorithms Executed by Each Processor

Algorithm A

Executed in response to: 'begin-phase-I' command issued by PPC.

Input arguments: None.

[We make the assumption that the A-memory of each processor can hold m records.]

Step 1: I=0. Nav=1. [Nav stores the next available memory location in the overflow area].
Step 2: I=I+1. If I=m+1, then go to Step 10. Else, read I-th record from A-memory.
Step 3: Check Condition Flag. If Condition Flag='Busy', then go to Step 3. Else, issue 'request-list-#' request to PPC with the following two arguments.

(1) The processor # of this processor.
(2) The join value of the record just read.

Step 4: Wait for 'request-accepted' signal from PPC. If this signal is not received, then go to Step 3. Else, go to Step 5.
Step 5: Receive list-# corresponding to the join value from PPC.
Step 6: Hash this list-# to an address H of a block of sequential B-memory.
Step 7: Check the Overflow Pointer of block H. If it is null, then go to Step 8. Else, set Ptr to Overflow Pointer and go to Step 9.
Step 8: Store the record in block H, if space is available in it, and go to Step 2. If no more space is available in block H, then store the record in the overflow area in location Nav. Store the value of Nav in the Overflow Pointer of block H, set Nav=Nav+1, and go to Step 2.
Step 9: Look at record pointed to by Ptr. If the value of the pointer associated with this record is non-null, set Ptr to this value and go to Step 9. Else, change the value of pointer associated with the record to Nav. Store, the record read from A-memory in Nav. Nav=Nav+1. Go to Step 2.
Step 10: Send 'phase-I-over' to PPC and terminate.

Algorithm B:

Executed in response to: 'begin-phase-II' command issued by PPC.

Input arguments: None.

[Assume that this algorithm is executed by processor j. Also, assume processor k follows processor j in the broadcast sequence.]

Step 1: I=0. Over=0 [Over is a variable that indicates if all records in a processor memory have been processed].
Step 2: Check the Flag Bit of processor j. [Flag Bit=1 indicates that a record exists in the Record Register which may have to be passed on to processor k]. If Flag Bit=1, then go to Step 3. If Flag Bit=0 and Over=0, then go to Step 7. Else, go to Step 2.
Step 3: Execute Algorithm Process using the record and the list-# in the Record Register as arguments.
Step 4: Increment Token by 1. [We may recall, from Figure 7, that Token is a part of the Record Register]. If Token=n, then go to Step 7. Else, go to Step 5.
Step 5: Execute Algorithm Send.
Step 6: Go to Step 2.
Step 7: I=I+1. If I=m+1, then go to Step 11. Else, read the I-th record from A-memory.
Step 8: Set Flag Bit of processor j to 1. Execute Algorithm Process using the record just read and the value 0 as arguments. If list-#=0, then set Flag Bit of processor j to 0 and go to Step 2.
Step 9: Execute Algorithm Send with the following two arguments:
(1) The record just read and its list-#.
(2) The value 1.
Step 10: Go to Step 2.
Step 11: Send 'phase-II-over' signal to PPC. Set Over=1. Go to Step 2.

Note: This algorithm will terminate on receipt of a 'Stop' signal from PPC. PPC sends this signal after it receives 'phase-II-over' signals from all the processors and it checks to see that the Flag Bits of all processors are reset.

Algorithm Process:

Executed in response to: Call from Algorithm B.

Input arguments: (a) The record to be processed.
(b) A list-#.

Step 1: If the list-# (Argument b) is not 0, then go to Step 5. Else, go to next step.
Step 2: Check Condition Flag. If Condition Flag='Busy', then go to Step 2. Else, issue a 'request-list-#' request to PPC with the following two arguments.

(1) The processor # j, of this processor.
(2) The join value of the record passed as argument to this algorithm.

Step 3: Wait for 'request-accepted' signal from PPC. If this signal is not received, then go to Step 2. Else, go to next step.
Step 4: Receive list-# corresponding to this join value from PPC. If list-#=0, then terminate. Else, go to next step.
Step 5: Hash this list-# to an address H of a block of sequential memory.
Step 6: Do Steps 7 and 8 for every record in the block and also every overflow record associated with this block.
Step 7: Check the join value of the record. If it is the same as the join value of the record passed on argument, then go to next step.
Step 8: Extract the required fields from the source record (the record in the block) and the target record (the record passed as argument). Send the results over to DBCCP which then routes them on to the front-end computer.
Step 9: Terminate.

Algorithm Send:

Executed in response to : A call from Algorithm B.

Input arguments: (1) The record and list-# to be sent.
(2) The value of the Token.

[Assume that the algorithm is executed by processor j and that processor k follows processor j in the broadcast sequence.]

Step 1: Check the Flag Bit of processor k. If it is 1, then go to Step 1. Else, go to Step 2.
Step 2: Place the first and second arguments in processor k's Record Register. Set processor k's Flag Bit to 1 and reset processor j's Flag Bit.
Step 3: Terminate.

We recall that these algorithms are executed the moment the first record is loaded into a processor. Therefore, simultaneous with the execution of these algorithms, other records are being placed in the memories of the processors via a direct memory access (DMA).

2.4 Analysis of Time Complexity

In order to make the calculations easier, the following realistic assumptions are made.

(1) A processor never finds PPC busy when it needs to issue a 'request-list-#' request. In the next chapter, we shall calculate the speed at which PPC will have to respond to such a request in order to make

this assumption realistic.

(2) A bucket can hold ten records. The average segment (bucket) size of a CCD memory is 2K bytes and a record is approximately 200 bytes, so that on the average ten records will fit a bucket.

(3) The memories are just large enough to hold all the records. That is, a load factor (ratio between the amount of memory used by stored records and the total amount of memory available) of 1 is assumed.

Let us try to calculate the average number of accesses that must be made before a record may be loaded (or retrieved). We shall make use of the results obtained from [8], where the formulas are given for the average number of overflow records and the average number of accesses required to retrieve a record when the overflow records from all buckets are stored in a common overflow area.

(1) $P(r)=e^{-m}m^{r}/r!$

where P(r) is the probability that r records are assigned to a bucket and m is the ratio of the number of records to be loaded and the number of buckets. With our assumption of a loading factor of one, m will be equal to the bucket size, i.e., 10.

(2) $Q(r)=\sum_{i=r}^{\infty}P(i)$.

(3) $\bar{i}(m,s)=sP(s) - (s-m)Q(s)$,

where s is the bucket size and $\bar{i}(m,s)$ gives the average number of records assigned to a bucket in excess of its capacity.

(4) $\bar{a}(m,s)=[\bar{i}(m,s-1) - (s-1)x\bar{i}(m,s)/m]/2$

where $\bar{a}(m,s)$ is the average number of additional accesses required, before a record may be loaded (or retrieved).

As we mentioned before, we are interested in $\bar{a}(10,10)$ and $\bar{i}(10,10)$. Thus,

$$\bar{a}(10,10)=[\bar{i}(10,9) - 9x\bar{i}(10,10)/10]/2$$

Since,

$$\bar{i}(10,9)=9P(9)+Q(9).$$

$$\bar{i}(10,10)=10P(10)=1.25$$

We have

$$\bar{a}(10,10)=[9P(9)+Q(9)-9/10x10P(10)]/2=[9P(9)+Q(9)-9P(10)]/2$$

But,

$$P(9)=e^{-10}10^{9}/9!$$

and,

$$p(10)=e^{-10}10^{10}/10!$$

and the two are equal so we have

$$\bar{a}(10,10)=[Q(9)]/2.$$

Q(9) is the probability that nine or more records will be assigned to a bucket. This is obviously less than one. Therefore, $\bar{a}(10,10)$ must be less than .5. Taking the worst possible case, we have $\bar{a}(10,10)=0.5$.

A. Phase-I Analysis:

Each processor has to do the following for each of its m records. Read the record, send join value to PPC, receive list-# from PPC, hash join value and store in appropriate place. We have already stated that we will ignore the time of communication with PPC. Also, since the hashing function is so simple, we will ignore the time taken to do the hashing. Let a be the time taken to access an arbitrary record (whether to read it or to store it in memory). Then, the time taken by a processor to execute Phase-I is

$$m(a + (1 + \bar{a}(10,10)a) = 2.5ma \quad .....I$$

Since the processors operate in parallel, this is the time taken to finish Phase-I.

B. Phase-II Analysis:

Let ℓ be the percentage of target records which participate in the join operation. Then, out of the mn target records, ℓmn of them will require

routing operations. On an average, each processor will be the originator of ℓm of these routing operations. A processor j will have to send a record on to processor k (where k follows j in the broadcast sequence) as long as the record did not originate from processor k. Thus, each processor will be involved in $\ell mn - \ell m = \ell m(n-1)$ routing operations. Also, for each target record, a processor must do the following. Read the record, send its join value to PPC, obtain a list-# from PPC, hash the join value to a block address and then examine all records in this block (and all overflow records of this block) for participation in the join operation. Once again, we ignore the time taken to communicate with PPC.

Let us assume that a routing operation takes s time units and that the time taken to read out an entire bucket of sequential memory is b time units. Since a processor is either sending records over to the next processor in sequence or examining its own records, the total time taken by a processor for Phase-II is

$$\ell m(n-1)xs + m(a+b+\bar{I}(10,10)xa) \quad \ldots\ldots \text{II}$$

Since each of the processors operate in parallel, this is the time taken to finish Phase-II.

By combining formulas I and II, we have that the entire time for the join operation is, therefore,

$$\ell m(n-1).s + m(3.5a + b + \bar{I}\ (10,10)xa)$$

Substituting $\bar{I}(10,10)=1.25$, we obtain the time for join.

$$\ell m(n-1)s + m(4.75a + b).$$

Thus, we see that the operation is of $O(mn)$.

2.5 Calculation of PPC Speed

We had earlier made the assumption that the processors never find PPC busy, so that the processors need not wait before issuing the 'request-list-#' request. In this section, we shall calculate the speed at which PPC

will have to respond to such requests in order to make the above assumption valid.

Naturally, we shall calculate the worst case behavior, that is, the fastest speed at which PPC will have to respond to such requests. We recall that selected records from the tracks of a cylinder are merged into a single stream. These records are then sent to the various processor memories. If PPC can respond to the requests at a faster rate than the inter-arrival rate of records, it will never be a bottleneck. The worst case is obviously when all records of a cylinder are selected for participation in the join operation. Let us make the following assumptions.

(1) The average record size is 200 bytes.

(2) The average track size is 30,000 bytes.

(3) The average number of tracks per cylinder is 20.

(4) The average speed of rotation of the disk device is 3600 rpm.

Now, the time taken for a single revolution of the device, i.e., to read 30,000 bytes, is (1/3600)x60x1000 = 16.67msecs. So, the time taken to read one record, i.e., to read 200 bytes, is (200/30000)x16.67x1000 = 111.1 μsecs. But, in 111.1 μsecs, 20 records are read (one from each track of the cylinder). So, a record is presented to the processor memories, one every 5.56 (=111.1/20) μsecs. This is the speed at which the PPC will have to respond to 'request-list-#' requests.

3. OTHER POST PROCESSING FUNCTIONS

We shall now consider some of the other post processing functions that will be carried out in the post processor (PP) of DBC.

3.1 Maxima and Minima

In many instances, a user is not interested in getting a set of records. Instead, he is often interested in finding the maximum (or minimum) value

taken by some attribute of a set of records. Consider, for example, a relational database consisting of the EMPLOYEE relation. Let us assume that many of the tuples in this relation carry information about clerks. A particular user may then like to find out the maximum (minimum) salary earned by a clerk. To satisfy the above request, DBCCP issues a 'retrieve-by-query' request to the mass memory (MM) using the following query:

$$(\text{Relation} = \text{EMPLOYEE}) \wedge (\text{Job} = \text{CLERK})$$

The retrieved records are then stored in PP which then proceeds to find the maximum (minimum) value taken by the Salary attribute of the retrieved records. The maximum (minimum) is found as follows.

First, each of the n processors makes one scan over all the records in its own memory and finds the record with maximum (minimum) value for the Salary attribute. Thus, each processor finds its own local maximum (minimum) and sends this local maximum (minimum) over to PPC. PPC may now find the overall maximum (minimum) from among these local maxima (minima). This overall maximum (minimum) is now routed to DBCCP which then returns it to the user via the front-end computer from which the query originated. Often, values of other attributes from the record which has the maximum (minimum) value for an attribute may be required. For example, a user may request the name of the clerk with the highest (lowest) salary.

To handle this case, a slight modification is made in the above algorithm. Once again, each processor finds a local maximum (minimum). However, it passes its own processor id along with the local maximum (minimum) to PPC. Also, each processor stores a pointer to this local maximum (minimum). PPC can now identify not only the overall maximum (minimum), but also the processor which contains the relevant record. PPC now asks the processor containing the relevant record to pass along the required information from this record to DBCCP. Since the processor will have a pointer to its local maximum (minimum), it

can easily satisfy the request.

The time for this algorithm is m+n, where m is the number of records per processor memory and n is the number of processors. This is because each processor can find a maximum (minimum) in m time units and PPC needs an additional n units to find the overall maximum (minimum).

3.2 Sum

Often, a user may request the sum of the values taken by an attrubute of a set of records. For example, consider a database containing records of fathers. One of the attributes in each record is the number of children (Numchild) that the father has. A user may require to know the total number of children in the community. To satisfy this request, it is necessary to take a sum of the values of the Numchild attribute from all the records. This is done as follows.

Each processor adds up the values of this attribute from all records in its memory and sends the local sum to PPC. PPC obtains all local sums and forms an overall sum which is then routed to DBCCP and thence to the front-end computer.

Once again, we see that the algorithm is of $\Theta(m+n)$, where m and n have the same meaning as before.

3.3 Average

Another request that is frequently made by users of a database involves finding the average of a set of attribute values. Thus, a user may request the average salary earned by a clerk. After all records pertaining to clerks have been retrieved by the mass memory, PP then proceeds to find the average of the Salary attribute as follows.

Each processor adds up the values of this attribute from all records in its memory and forms a resultant local sum. Each processor also counts the number of records in its memory and sends both the local sum and the number of

records in its memory to PPC. More formally, each processor executes the following algorithm:

Step 1: Sum=0; Count=0.
Step 2: Read next record. If no more records, then go to Step 4. Else, go to Step 3.
Step 3: Count=Count+1. Sum=Sum + value of relevant attribute of the retrieved record. Go to Step 2.
Step 4: Send Sum and Count to PPC and terminate.

PPC now finds the sum of all the local sums and makes an overall sum. Additionally, it finds the total number of records by adding up the number of records in each processor's memory. Thus, it can form an average value, which is then routed to DBCCP.

The complexity of this algorithm is also O(m+n).

3.4 Count

Another set operator that would be useful if incorporated into DBC is the Count operator. It merely makes a count of the number of records selected from MM by means of a query. Thus, a user may wish to know the number of clerks with salary greater than $500. Let us assume that the EMPLOYEE relation stores all records concerning information about clerks. In response to this request, DBCCP first issues a 'retrieve-by-query' request using the following query:

((Relation=EMPLOYEE) ∧ (Job=CLERK) ∧ (Salary > 500)).

The retrieved records are than placed in the memories of PP which then proceeds to count the number of records as follows.

Each processor counts the number of records in its memory and sends this local count to PPC. PPC now adds up all the local counts to come up with a figure for an overall count of the number of records.

It is easily seen that the algorithm is of O(m+n), where m and n have their usual meaning.

3.5 Set Inclusion Operator

This operator will decide, given a value, if it is one of a set of given values. Thus, a user may be interested in employees who stay in Dayton, Columbus or Cleveland. One way to retrieve the relevant records would be to issue to MM, the following query:

((Relation=EMPLOYEE) ∧ (Location=DAYTON))

∨ ((Relation=EMPLOYEE) ∧ (Location=COLUMBUS))

∨ ((Relation=EMPLOYEE) ∧ ((Location=CLEVELAND))

The above query consists of three conjunctions. We may recall, from [3], that the mass memory takes one disk revolution to evaluate each conjunction. Let us assume that there is a record with Location=CINCINNATI. When this record comes under the read head the first time, the value of the Location attribute is compared to DAYTON and is not compared to COLUMBUS or CLEVELAND, since, these values are in the next conjunctions. Thus, the comparison of the value of the Location attribute to COLUMBUS must be made in a second revolution and the comparison to CLEVELAND must be made in a third revolution. Hence, three revolutions of the disk device are necessary to evaluate the above query.

Alternatively, the following method may be employed. First, all records that satisfy the query (Relation=EMPLOYEE) are retrieved and placed in PP's memories. Then, PP invokes the set inclusion algorithm to retrieve the relevant records. PP operates as follows.

The set of values against which each record must be compared is stored in the first fields of AM of PPC. Each processor does the following. It reads a record, extracts the value of the relevant attribute (in this case Location) and sends it over to PPC. PPC looks through its AM for a match. If there is a match, then the value is one of the values in the given set, and PPC returns a

positive signal; otherwise, PPC returns a negative signal. On receipt of a positive signal, the processor routes the record just read to DBCCP and then reads the next record from its memory. On receipt of a negative signal, the processor discards the record and then proceeds to read the next record from its memory.

The reader may feel that PPC would tend to become a bottleneck in the execution of this algorithm. However, if PPC can handle n requests (n is the number of processors in PP) in the time taken to retrieve a record from processor memory, it will not be a bottleneck. Since the memories used in PP will have access times of about 1ms, and since we expect to have n=16 processors, PPC has to respond at a speed of (1000/16)=62.5 μsecs. Even with 32 processors PPC response time only needs to be 31.25 μsecs. Earlier, we had calculated that PPC must operate at 5.56 μsecs. Hence, PPC will not be a bottleneck in the above algorithm. Also, the entire algorithm will only take m time units, where m is the number of records per processor.

We note, that this method forces us to discard many records that are retrieved from the mass memory. However, this method will have a distinct speed advantage over the earlier method which involves as many disk revolutions as there are conjunctions in the user's query -- especially if there are many conjunctions, i.e., the set of values to be searched is very large.

4. CONCLUSION

We have proposed a PP architecture involving n processors with sequential memory and log n interconnections among processors, and shown how to carry out various post processing functions such as relational join, maxima, minima, average, sum, and count. We have also shown how to improve DBC's response to certain kinds of queries by the introduction of a set inclusion operator.

REFERENCES

[1] Banjerjee, J., Baum, R., and Hsiao, D. K., "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, Vol. 3, No. 4, Dec. 1978, pp. 347-384. Also available in, Baum, R. I., Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part I: Concepts and Capabilities," Technical Report OSU - CISRC - TR - 76 - 1, The Ohio State University, Columbus, Ohio, Sept. 1976.

[2] Kannan, K., Hsiao, D. K. and Kerr, D. S., "A Microprogrammed Keyword Transformation Unit for a Database Computer," Proceedings of the Tenth Annual Workshop on Microprogramming, Oct. 1977, Niagara Falls, New York, pp. 71-79; and Hsiao, D. K., Kannan, K., and Kerr, D. S., "Structure of Memory Designs for a Database Computer, "Proceedings of ACM 77 Conference, Oct. 1977, Seattle, Washington, pp. 343-350; Also available in Hsiao, D. K., and Kannan, K., "The Architecture of a Database Computer -- Part II: The Design of the Structure Memory and its Related Processors," Technical Report OSU - CISRC - TR - 76 - 2, The Ohio State University, Columbus, Ohio, Oct. 1976.

[3] Kannan, K., "The Design of a Mass Memory for a Database Computer," Proceedings of the Fifth Annual Symposium on Computer Architecture, April 1978, Palo Alto, California, pp. 44-50; Also available in Hsiao, D. K., and Kannan, K., "The Architecture of a Database Computer -- Part III: The Design of the Mass Memory and its Related Processors," Technical Report OSU - CISRC - TR - 76 - 3, The Ohio State University, Columbis, Ohio, Dec. 1976.

[4] Banerjee, J., Hsiao, D. K., and Menon, J. M., "The Clustering and Security Mechanisms of a Database Computer (DBC)," Technical Report OSU - CISRC - TR - 79 - 2, The Ohio State University, Columbus, Ohio, April 1979.

[5] Banerjee, J., and Hsiao, D. K., "Parallel Bitonic Record Sort - An Effective Algorithm for the Realization of a Post Processor", Technical Report OSU - CISRC - TR - 79 - 1, The Ohio State University, Columbus, Ohio, April, 1979.

[6] Copeland, G. P., Lipovski, G. J., and Su, S. Y. W., "The Architecture of CASSM: A Cellular System for Non-Numeric Processing," Proceedings of the First Annual Symposium on Computer Architecture, Dec. 73, pp. 121-128.

[7] Su, S. Y. W., and Emam, A., "CASDAL: CASSM's Data Language," ACM Transactions on Database Systems, Vol. 3, No. 1, March 1978, pp. 57-91.

[8] Van der Pool, J. A., "Optimum Storage Allocation for Initial Loading of a File", IBM Journal of Research and Development, Nov. 1972, pp. 579-586.

COMPUTER &
INFORMATION
SCIENCE
RESEARCH CENTER